Design Exploration in Hw/Sw Co-design of Real-Time Object-oriented Embedded Systems: the Scheduler Object

Elias Teodoro Silva Jr1,2; Marco A. Wehrmeister1; Leandro Buss Becker3; Flávio Rech Wagner1; Carlos Eduardo Pereira4

1 Computer Science Inst., Federal University of Rio Grande do Sul, Brazil, {etsilvajr, maweehrmeister, flavio}@inf.ufrgs.br
2 on leave from Telecommunication and Computer Department, Federal Center of Technological Education of Ceará, Brazil
3 Automation and Control Systems Department, Federal University of Santa Catarina, Brazil, lbecker@das.ufsc.br
4 Electrical Engineering Department, Federal University of Rio Grande do Sul, Brazil, cpereira@eletro.ufrgs.br

Abstract

This paper discusses a design flow for multithread object-oriented real-time applications, running on top of an embedded, platform-based, customizable Java processor, which is prototyped using affordable FPGAs. The proposed approach enforces design space exploration activities, taking into account aspects like temporal behavior, memory footprint, and power/energy consumption. A case study containing a task scheduler implementation as both software and hardware modules is presented. While both implementations are compatible with the developed program from an interface point of view, they lead to different timing and footprint requirements. Their evaluation in terms of memory occupation and number of FPGA logic cells is presented.

1. Introduction

Real-Time systems depend not only on the logical results of computation, but also on the timeliness of those results [5] [6]. Most real-time systems are special-purpose and complex, require a high degree of fault tolerance, and are typically embedded in a larger system. They also include substantial knowledge about the application and its environment, since they are characterized by a high coupling between a computer-based system (a controlling system) and a technical process (or the controlled system). Examples include industrial automation, flexible manufacturing systems, process control and flight control, among others.

Embedded real-time systems are becoming larger and more complex, demanding development tools that work at higher abstraction levels. In order to cope with time-to-market requirements, developers look for reusable components (both hardware and software). In this respect, the designer must have the opportunity to trade off between software (for flexibility) and hardware (for performance and low power) implementations of components and sub-systems.

The SEEP project (Platform-Based Design of Embedded Systems) that is under development at the Federal University of Rio Grande do Sul proposes a new methodology for embedded real-time systems design. The project offers a complete toolset for embedded systems modeling, analysis, validation, and synthesis. This allows fast design space exploration in the hardware and software co-design of embedded real-time systems. The approach is based on the reuse of hardware and software components and on the configuration of architectural platforms implemented upon programmable devices such as FPGAs.

This paper illustrates the design space exploration aspect of the design methodology underlying the SEEP project. As an example, the paper focus on the implementation of the “scheduler” object for a real-time embedded system, as defined by the Real-Time Specification for Java (RTSJ) [4]. This case study provides design space exploration regarding the scheduler implementation in software or hardware,
with both keeping the same high-level object model as described along the paper.

The remaining parts of the paper are organized in the following way. Section 2 gives a brief overview of the SEEP methodology. Section 3 introduces the hardware and software components used to develop object-oriented real-time embedded applications, which includes the FemtoJava processor, a customizable RT-Java processor. Section 4 discusses the scheduler implementation in hardware and in software. Section 5 presents a case study that allows a comparison between the two implementations of the scheduler for a dedicated embedded system, in terms of performance and cost. Finally, Section 6 draws the main conclusions of the paper.

2. SEEP Methodology Overview

The SEEP project proposes a complete and integrated methodology for the design and test of embedded real-time systems, taking into account non-functional requirements such as temporal behavior and power consumption. The SEEP methodology makes extensive use of object-oriented concepts, from system specification to its implementation, aiming at obtaining reusable system components for which different implementations, meeting different design trade-offs, are available. The methodology supports designers in a rapid, but cost-effective, design space exploration.

The SEEP methodology is depicted in Figure 1. It starts with the definition and further validation of a high-level object-oriented model, which follows the UML-RT profile [14]. Problem domain objects are identified and class hierarchies mapping concepts and semantics of the application domain are described. Then, a system exploration step follows, allowing the designer to decide on major functional trade-offs, for instance by selecting the right algorithms for implementing the desired functionality. Then, architectural design space exploration is performed, where the designer will decide on major architectural blocks for the system implementation, assigning functions to architectural components. This exploration is based on already available platforms and on a library of IP (“intellectual property”) components. The result is an abstract macro-architecture, which implements a hardware-software partitioning, where some functions are mapped to software tasks and other ones to dedicated hardware blocks. This high-level architectural model still abstracts all low-level implementation details. Two platforms are being currently considered in the SEEP project. The first one is based on a family of Java processors (see Section 3.1), implemented on conventional FPGAs, while the second one is a commercial platform, containing a large FPGA with two embedded PowerPC processors.

![Figure 1 – The SEEP design methodology](image)

Hardware and software synthesis follows in the development cycle. As part of the software synthesis process, an operating system may be required to implement services like inter-process communication and task scheduling. This operating system must be also dedicated to the application and optimized for the given design requirements. A VHDL description of a cycle-and-pin accurate micro-architecture is generated for the hardware part. Inputs to this task are VHDL descriptions of the selected architectural blocks, already available in the component library. These descriptions may be parameterized, and their configuration depends on design decisions taken at earlier design exploration steps.

3. Configurable RT-Java Processor

3.1 FemtoJava Processor

FemtoJava [10] is a stack-based microcontroller that natively executes Java bytecodes, whose major characteristics are a reduced and configurable instruction set, Harvard architecture, and small size. It implements an execution engine for Java in hardware, through a stack machine compatible with the specification of the Java Virtual Machine (JVM). A
compiler that follows the JVM specification is being used and allows the synthesis of an ASIP (application-specific integrated processor) version of FemtoJava. Only the execution core and some tools to extract the software at design time are really necessary.

An immediate advantage of native execution of Java bytecodes is software compatibility. It guarantees the availability of cross-platform software development. In some conventional Java platforms (for example, PCs or workstations) one can just implement and run a Java program. Running the program is equivalent to simulating the behavior of the application in the target microcontroller, with all resources and convenience of a desktop environment in the development phase.

The basic FemtoJava implementation used a subset of the JVM bytecodes, with only 68 instructions. The supported instructions were basic integer arithmetic and bitwise operations, conditional and unconditional jumps, load/store instructions, stack operations, and two extra bytecodes for arbitrary load/store. In this core, all instructions are executed in 3, 4, 7, or 14 cycles, because the microcontroller is cacheless and several instructions are memory bound. In order to support multithread applications, the instruction set of FemtoJava was expanded [8] [9]. Enhancements in performance are obtained with pipelined and VLIW versions of the processor [13].

3.2 The Real-time API

The RTSJ defines an API for the Java language that allows the creation, verification, analysis, execution, and management of real-time threads, whose correctness also depends on the fulfillment of timing requirements. Unfortunately, the popularization and use of this API is still incipient given the limited availability of JVMs that conform to this specification. On the other hand, the situation is different in the embedded systems domain, since there are several proposals that optimize the JVM to embedded targets. Such optimizations aim at reducing the required footprint for a JVM as well as to improve the application performance. The Sashimi environment [10] is an example of JVM optimization for embedded systems. It provides a powerful and easy-to-use development environment for embedded systems that has been successfully applied to different case studies. The Sashimi environment was recently extended to incorporate an API that supports the object-oriented specification of concurrent tasks and allows the specification of timing constraints [9]. Those facilities increase the code abstraction level and optimize the development of real-time embedded systems. Figure 2 shows the UML model of the API used in this work, which is a subset of the RTSJ specification. This API offers an alternative to simplify the design of a real-time object-oriented application running on a Java native code processor. Using this API together with the Sashimi environment, programmers can design concurrent real-time applications and make their synthesis into the FemtoJava processor.

Figure 2 – RTSJ-based API

The API uses the concept of schedulable objects, which are instances of classes that implement the Schedulable interface, for instance the RealtimeThread. It also uses a set of classes to store parameters that represent a particular resource demand from one or more schedulable objects. The ReleaseParameters class (superclass from AperiodicParameters and PeriodicParameters), for example, includes several useful parameters for the specification of real-time requirements. Moreover, it supports the expression of the following elements: absolute and relative time values, timers, periodic and aperiodic tasks, and scheduling policies. The term ‘task’ derives from the scheduling literature, representing a schedulable element within the system context. It is also a synonym for schedulable object.

In order to provide a framework with different scheduling policies, a new scheduling structure for the FemtoJava processor has been designed. The adopted approach allows designers to program their algorithms using higher levels of abstraction. More specifically, they can make use of the Java programming language together with the provided API. The intent is to minimize architecture-dependent characteristics within
the scheduling algorithms, making the framework as general as possible.

4. Alternative Implementations for the Scheduler Object

4.1 Scheduler Object in Software

When implemented in software the scheduler object consists of an additional runtime process (or task) that is in charge of allocating the CPU for those application-processes, which are ready to execute, exactly like in any RTOS. Application developers should choose the most suitable scheduling algorithm at design time. Later on, this algorithm is synthesized with the scheduler process into the embedded target.

The scheduling framework provides efficient dispatching and context switching mechanisms that are able to work with any fixed-priority algorithm, like Rate Monotonic (RM) [3]. Additionally it includes support for dynamic scheduling with the EDF (Earliest Deadline First) algorithm. Therefore, the framework can support both static and dynamic scheduling algorithms.

4.2 Scheduler Object in Hardware

The hardware scheduler object has the same responsibilities. However, its hardware component contains additional tables that store task descriptors sent by the FemtoJava processor, as well as operators to manipulate those tables. A class called HardwareScheduler, which interacts with the real hardware and performs context switching and dispatching, encapsulates the hardware. Context switching and dispatching imply a minimum cost when compared to the scheduling computation, especially when using complex scheduling algorithms.

By moving the scheduling algorithm from software to hardware, this operating system function no longer competes with the application tasks for the processor. Now, the scheduling function has its own dedicated hardware unit, which is able to: i) run more complex scheduling algorithms; and ii) provide a really non-intrusive task scheduling, thus enhancing the tasks temporal predictability [2]. Additionally, the scheduler coprocessor can be used to obtain accurate execution time measurements for the real-time tasks being executed, which is an important parameter to perform schedulability analysis.

4.2.1 Architecture

The proposed architecture for the hardware scheduler is shown in Figure 3. The main components of this scheduling coprocessor are the following:
- General Register block,
- Scheduler block,
- SyncEvent block, and
- AsyncEvent block.

The General Register block contains all task descriptors independently of their state (running, blocked, ready, idle). At any given moment, the main processor may include or remove tasks in this block, which should confirm or refuse new tasks.

The AsyncEvent block receives tasks that will be event-triggered and monitors external triggering events whose instant of occurrence cannot be defined a priori. These events or signals are typically associated to sensors, such as an alarm occurrence or a door opening. When an event arrives, the related task ID is sent to the Scheduler block.

The SyncEvent block is similar to the AsyncEvent block, however it monitors the application time-triggered tasks, which can be periodic or one-shot tasks. The execution of this kind of task depends on the time tick provided by the real-time clock.

The Scheduler block receives tasks that are ready to execute, sent by SyncEvent or AsyncEvent, and put them into its table in the correct order according to the established schedule policy. The Scheduler sends tasks to the processor in two situations:

1) The task currently running finishes its execution. The processor performs a read operation and receives the next task from the coprocessor.

2) The task recently inserted on the first position of the Scheduler block table has a higher priority than the task currently executing in the processor. The
scheduling coprocessor interrupts the FemtoJava processor, preempting the running task.

4.2.2 Hardware scheduler commands

To configure the hardware scheduler, the FemtoJava processor can use five different commands:
- Include a task – The coprocessor receives a set of parameters that describes the task to manage. Figure 4 shows the details of these parameters.
- Remove a task – The coprocessor removes a previously included task.
- Start a task – The coprocessor switches a previously programmed (included) task to the Active state and can schedule this task when its event arrives.
- Task finished – The coprocessor supplies the ID of the first task in the ready task queue.
- Preempt task – The coprocessor supplies the ID of the first task in the ready task queue and reschedules the running task.

As shown later, all these operations are transparent to the designer who can use only the API, choosing the hardware or the software implementation of the scheduler.

Figure 4 – Sequence of task programming parameters for a priority-based scheduling algorithm

4.2.3 Hardware scheduler interface

FemtoJava communicates with the coprocessor and I/O devices through a 32-bit bus. Since they are both synthesized on the same chip, the communication overhead is minimized. The main effort of the communication system is during the start-up process, which usually does not have time constraints. For each task, a data block similar to that one shown in Figure 4 is sent from FemtoJava to the coprocessor. In this example, the set of seven 32-bit parameters is used to describe a synchronous task in a priority-based scheduling algorithm. The first word is a control-word, which has the following fields:
- St_Priority – Static priority of the task.
- EV – Defines which external input will start the task (only for asynchronous tasks).
- ID – Task identifier.
- Sync_or_Async – Defines if this is a synchronous (Periodic or one-shot) or an asynchronous (sporadic) task.
- Start – Indicates when the task should be activated (if synchronous).
- Period – Indicates the activation period of the task (if periodic).

5. Design Space Exploration for Scheduler Object

This section illustrates the use of the proposed API in an embedded system design, highlighting software and hardware implementation aspects of the scheduler. The application to be implemented consists of a synthetic workload that simulates realistic applications. The use of hardware and software scheduler versions is shown in two implementation examples.

5.1 Task Implementation

In order to illustrate the design, two hypothetical tasks are used, Task1 and Task2. The main class is named TaskTest, which is responsible for object allocation, initialization, and starting (applied for the real-time tasks). The main parts of its code are shown in Figure 5. The initSystem() method represents the starting point of the application execution flow. The real-time tasks are attached to the scheduler by the addToFeasibility() method and their startup can be identified by the call to the start() method. The idleTask() method is called when there is no task ready to run. More details about the code can be found in [9]. The code of one task, which instance is named t1, in the TaskTest class, is shown at Figure 6. In the example, Task1 is started each 25ms.

5.2 Hardware Implementation

The implementation of the scheduler coprocessor was described in VHDL. Currently, a fixed-priority scheduling algorithm is available for the hardware scheduler. Future work will implement other scheduling algorithms and evaluate their cost and performance.
public class TaskTest {
    // static allocations
    public static PriorityScheduler mySched
        = new PriorityScheduler();
    public static void initSystem() {
        Scheduler.setDefaultScheduler(mySched);
        t1.addToFeasibility();
        t2.addToFeasibility();
        t1.start();
        t2.start();
        idleTask();
    }
}

Figure 5 – Main class for the example application using software scheduler.

public class Task1 extends RealtimeThread {
    // constructors
    public void mainTask() {
        while (m_SchedCount < 15) {
            m_SchedCount++;
            // ... Repetitive operations
            waitForNextPeriod();
        }
        finish();
    }
    public void exceptionTask() {
        // handle deadline missing
    }
}

Figure 6 – Task class for the example application.

public class TaskTest {
    // idem
    public static HardwareScheduler mySched
        = new HardwareScheduler();
    public static void initSystem() {
        // idem
    }
}

Figure 7 – Main class for the example application using hardware.

In order to use the hardware scheduler, the developer follows the same methodology as explained in Section 5.1. Using the example shown in Figures 5 and 6, the application can be completely reused. The single required modification is the scheduler class used, as highlighted in Figure 7. Since the coprocessor only runs the scheduling action, context switching and dispatching are still executed in software, in the HardwareScheduler Class.

6. Experimental Results

For the experimental investigations the design was restricted. The time definitions were reduced to a single 32-bit field, representing only milliseconds, although in the initial models and in Java it is possible to represent time in three 32-bit fields (days, milliseconds, and nanoseconds). Moreover, the AsyncEvent block was removed to simplify the tests. All the tasks considered in the experiments are synchronous and cyclic (periodic) tasks.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>15181</td>
<td>3305</td>
</tr>
<tr>
<td>GeneralRegister</td>
<td>571</td>
<td>302</td>
</tr>
<tr>
<td>Converter</td>
<td>87</td>
<td>85</td>
</tr>
<tr>
<td>SyncEvent</td>
<td>11738</td>
<td>2185</td>
</tr>
<tr>
<td>RTC</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>Scheduler</td>
<td>2690</td>
<td>638</td>
</tr>
</tbody>
</table>

Table 1 – Number of logic cells used by coprocessor

The VHDL model was compiled using Altera tools - Quartus II 4.0 Web Edition Full. The description used latches to implement the data tables of subsystems (blocks). For the synthesis, the Cyclone device family was applied. Table 1 shows the area, in logic cells, of each block, for synthesis with 4 and 8 tasks. The coprocessor has a relatively large area, since the FemtoJava processor costs nearly 3500 logic cells. This area overhead is the price to be paid for a low time overhead and low energy, as shown subsequently. The RTC block is the system real-time clock, and the Converter block is a state machine that receives words from the processor and formats them for dispatching to GeneralRegister block.

A coprocessor model, described in C language, was simulated using modified versions of SASHIMI and CACO-PS [1]. Applications with 2, 4 and 8 tasks, running 15 times, were evaluated. The tasks are synthetic loads, which simulate real applications. A 500 MHz clock was adopted in these simulations.

In order to verify the time properties of the scheduler, the latency of the higher priority task was measured. This is the time since the instant when the task was programmed to start until the instant when it effectively starts. Figure 8 shows this latency for experiments with 2 and 8 tasks. It is clear that the delay time to start the task is very short. Even more, it is almost constant and independent of the number of tasks running. Since context switching and dispatching have fixed costs and the scheduling is in hardware, the number of task does not affect the scheduling performance. Naturally, as the number of tasks grows, the coprocessor will use more logic cells (Table 1).

A software version of the scheduler was implemented and evaluated with the same benchmarks used in hardware scheduler. Only the processor frequency was increased to 5 MHz to meet deadlines...
of the tasks, since now the scheduler consumes a considerable time. Increasing processor frequency represents larger energy consumption. Although the introduction of coprocessor also implies more energy use, the final result is more energy effective. In order to show the time expended to switch between two subsequent tasks, the graph of the Figure 9 was built. This picture shows the cost to process the scheduling algorithm in the software versions versus the hardware version, for the first 15 executions. The hardware scheduler is faster because it manipulates its tables in a few clock cycles, while the software scheduler accomplish tens of instructions. In spite of this, the software scheduler could be an interesting solution when there is no free space in the chip. Since there are no system tasks running concurrently to application tasks, all occurrence of the scheduler have the same cost.

7. Conclusions and Future Works

The paper presented how developers can use the same API to implement a multithread object-oriented real-time embedded application, choosing hardware- or software-implemented schedulers. Latencies, for both implementations, and area, for hardware version, were evaluated.

The application development process is not influenced by architectural choices and does not need to consider if the scheduler must be implemented in hardware or software. Moreover, the developer can reuse all the application code if he/she needs to change the scheduler implementation, as long as he/she uses the proposed methodology.

The hardware scheduler reduces the latencies and is not affected by the number of tasks running, besides improving the predictability of the system, which is highly desirable for real-time applications.

The main cost imposed by the scheduler coprocessor is the area. In the example with 8 tasks, its area is larger than the FemtoJava processor area.

The main principle investigated here, which is the encapsulation of a hardware facility, will be explored for other system functions, like inter-process and even inter-processor communication, increasing reusability and reducing costs. The authors also intend to implement more complex scheduling algorithms, like TAFT (Time-Aware Fault-Tolerant) [15].

Acknowledgments

Thanks are given to the Brazilian funding agency CNPq, which is the project sponsor. The authors also thank the other SEEP-project researchers for the valuable discussions.

References